Please note:

If you experience some difficulty in viewing some of the pages, use the magnifying tool to enlarge the specific section

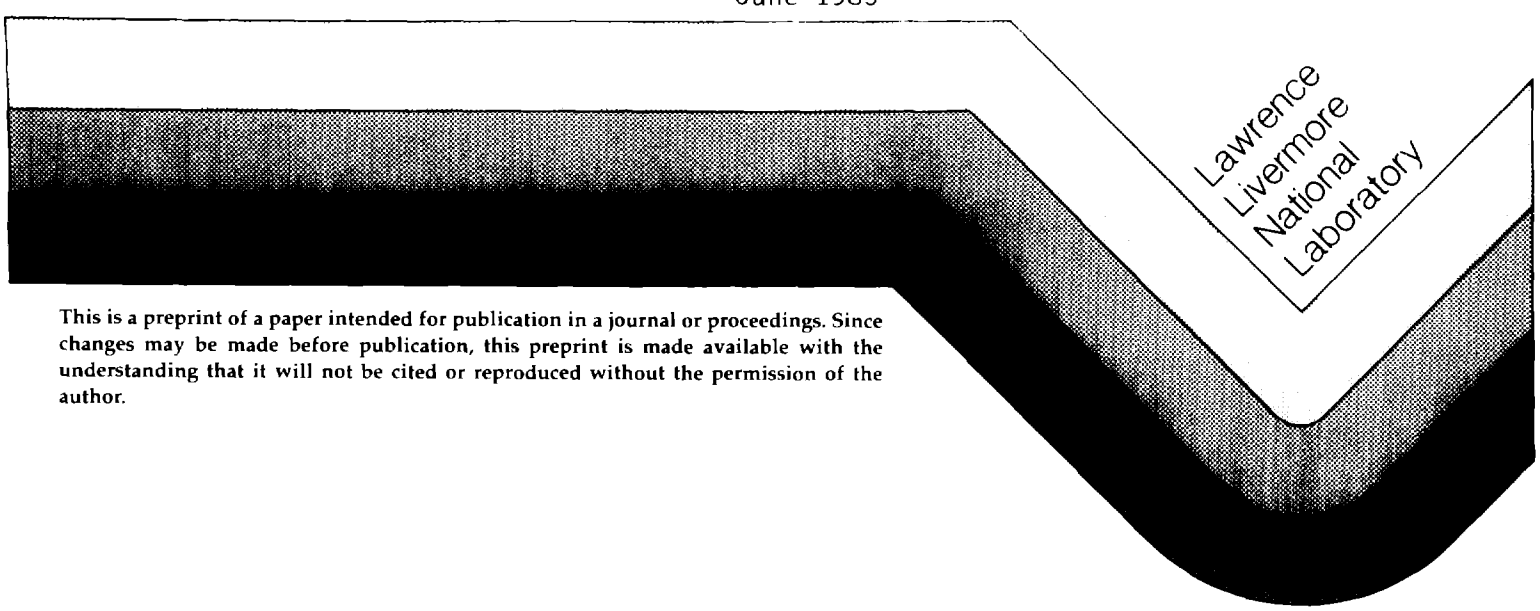
UCRL-92867
PREPRINT

RESULTS AND COMPARISONS IN MULTIPROCESSING USING VMS 4.0 AND MA780

N. E. Werner

This paper was prepared for submittal to
Digital Equipment Computer Users Society
Spring 1985,
New Orleans, LA,
May 27-31, 1985

June 1985



RESULTS AND COMPARISONS IN MULTIPROCESSING USING VMS 4.0 AND MA780*

Nancy E. Werner
Lawrence Livermore National Laboratory
P.O. Box 808, MS L306
Livermore, CA 94550

Abstract

Experiments using different parallel processing techniques on selected parallel algorithms were performed. Relative performance of these techniques was observed. The hardware was 4 clustered Vax-780s with 14 to 16 Megabytes each of local memory and 4 Megabytes of shared memory (2 MA780s).

INTRODUCTION

Parallel processing is the ART of doing multitasking on more than one processor, where multitasking is the splitting up of a Job into many separate Tasks. Normally these Tasks need to communicate with each other in order to complete the Job. In a tightly coupled system, they will use Shared Memory for communication. In a loosely coupled system, they will send messages to each other via a common bus such as the CI Bus. With the present hardware, four clustered Vax-780s with 16 Megabytes, 14 Megabytes, 14 Megabytes, 14 Megabytes local memory respectively and 4 Megabytes shared memory, either method of communication could be used. Only the tightly coupled method has been pursued so far and will be discussed here.

MOTIVATION

Lawrence Livermore National Laboratory (LLNL) also has on site a four processor CRAY computer, the XMP-48. It would be nice if users could become familiar with parallel processing on a cheaper, friendlier and more accessible environment than is yet offered on the CRAY. The Vax System's main purpose is for parallel processing research; there are no production jobs to worry about, and dynamic debugging tools are available.

Most potential users at LLNL are not familiar with VMS. In order to lure them onto the VAX System, it was necessary to imitate the environment of the CRAY as much as possible. CRAY users were using a set of primitives devised by Cray Research Inc. (CRI) which were referred to as the "CRI Multitasking Primitives"[1]. These are simply a library of routines which were designed to be used for implementing parallel processing algorithms. A similar library was implemented on the VAX System to be as consistent as possible with the CRI library [2]. Programs which run on the CRAY, with minor modifications, can also run on the VAX System, within memory size limitations. Programs have been debugged on the VAX System and then successfully moved onto the CRAY.

PARALLEL PROCESSING DEFINITIONS

There are some basic things one must do for parallel processing that are not necessary for sequential processing. It must be possible to define Tasks which can execute on the available processors. The consistency of the shared data must be assured; simultaneous updates to the same data must be avoided. A section of code that alters shared data must be executed by only one processor at a time; such a section of code is called a **Critical Section.** The Tasks often must synchronize their activities with each other. A place at which Tasks need to meet before proceeding with the computation is called a **Barrier.**

A **Logical Processor** is a process which has been initiated at Job submittal time and is scheduled by the Operating System on the VAX on which that process resides. A **TASK** is an instantiation by a **Logical Processor** of a subroutine call with shared memory arguments. When the **TASK** has completed (returned), the **Logical Processor** is free to activate another **TASK.**

To implement a **Critical Section** , a **LOCK** can be used. A **LOCK** is a resource protector; only one **TASK** at a time is allowed to have a specified **LOCK.** If a **LOCK** is gotten before entering a section of code , then anyone else attempting to get that **LOCK** must wait until it is released. When the **Critical Section** of code has been completely executed, the **TASK** should then release the **LOCK** to allow another **TASK** which has also requested this **LOCK** to proceed. The same **LOCK** should be used for related **Critical Sections** which affect the same data.

Barriers can be implemented using **EVENT**s and/or **LOCK**s. An **EVENT** is a system wide signal that can be set, tested and cleared by all **TASK**s working on this Job. There are many ways to implement a **Barrier.** One example is given in Appendix B.

PARALLEL PROCESSING LIBRARY

The library implemented for VMS contains not only those subroutines as defined by CRI, but also by

* This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore under Contract No. W-7405-Eng-48.

necessity ,some special subroutines which must be used to map and unmap predefined areas of data to the shared memory. To facilitate this mapping, the user must place all his shared data into a common block named /SHAREDGLOBAL/. The library will also map some of its own data to shared memory. Other subroutines were added to provide more functionality.

DATA DEFINITIONS

The data was set up to be compatible with the CRI definitions as far as possible. Please note that an integer is 32 bits on the VAX and 64 bits on the CRAY.

Taskarray: 2-3 integers used to hold TASK information as follows:

count: number of integers in this array (set by user)

pointer: pointer to library data for this TASK (used only by library)

Taskvalue: optional value associated with this Task (set by user)

Name: subroutine name (entry point for TASK instantiation)

List: argument list for subroutine (addresses of arguments {must be in shared memory})

Taskvalue: user defined value (32 bits)

Lockdata: integer used to represent a LOCK (defined by user, manipulated by library)

Eventdata: integer used to represent an EVENT (defined by user, manipulated by library)

First_shr: address of the first data item in shared memory
(%loc(*) where common/sharedglobal/*..../)

Last_shr: address of the last data item in shared memory
(%loc(*) where common/sharedglobal/....*/)

Dbug: integer flag which will cause library to write tasking information to the log file if > 0

SPECIAL SUBROUTINES

Task_init (First_shr, Last_shr, Dbug)

Subroutine which maps the shared data to shared memory. It must be called before any shared data is used. It also sets up the Task_cleanup subroutine as an exit handler.

Task_cleanup

Subroutine which unmaps the shared memory.

CRI COMPATIBLE SUBROUTINES

Tskstart (Taskarray, Name [,List])

Startup the TASK associated with Taskarray by calling subroutine Name with arguments List.

TskWait (Taskarray)

Wait for the TASK associated with Taskarray to complete.

Logical = **Tsktest** (Taskarray)

If the TASK associated with Taskarray exists, then set Logical = .true.

TskValue (Taskvalue)

Retrieve the value for this TASK.

Lockasgn (Lockdata)

Assign and initialize the LOCK associated with Lockdata.

Lockrel (Lockdata)

If there are no waiters release the LOCK associated with Lockdata

otherwise set error condition.

Lockon (Lockdata)

If the LOCK associated with Lockdata is busy then wait,

otherwise get the LOCK.

Lockoff (Lockdata)

Relinquish the LOCK associated with Lockdata.

Logical = **Locktest** (Lockdata)

If the LOCK associated with Lockdata was already on,
set Logical = .true. and return

otherwise get the LOCK and
set Logical = .false.

Evasgn (Eventdata)

Assign and initialize the EVENT associated with Eventdata.

Evrel (Eventdata)

If there are no waiters, release the EVENT associated with Eventdata

otherwise set error condition.

Evpost (Eventdata)

Post the EVENT associated with Eventdata.

Evclear (Eventdata)

Clear the EVENT associated with Eventdata.

Evwait (Eventdata)

Wait for the EVENT associated with Eventdata to be posted.

Logical = **Evtest** (Eventdata)

If the EVENT associated with Eventdata was posted, Logical = .true.

otherwise Logical = .false.

ADDITIONAL SUBROUTINES

These subroutines can be used to obtain a chronological log of what is happening during a Job. If the Job is running on more than one VAX , the time is not a correct indicator since each VAX has a separate clock and they are not synchronized.

Open_shared (Unit, Filename, Record_size)

The file Filename is opened as a shared relative file with maximum record size = Record_ size and associated with Unit. If being called by Fortran, Unit is also the unit number.

Close_shared (Unit)

The file associated with Unit is closed and reset.

Get_nxtrec (Unit, Record_number, Count)

The next Count records are reserved on the file associated with Unit and the first of these record numbers is returned to Record_number.

Another set of subroutines was implemented to facilitate dynamic partitioning of the Job's work among the TASKS. A unique set of mailboxes may be set up with specified message size with which TASKS may communicate. Dynamic partitioning is achieved by dividing the work up and then putting the pieces into a mailbox queue; each TASK that is doing that work can then retrieve pieces of the work until the work is completed and the mailbox is empty.

Setup_sr (Mbx_array, Count, Mbx_ size, Code)

Set up Count mailboxes whose message size is Mbx_size and whose unique identifier will be Code. The channel number for each mailbox initialized will be placed in Mbx_array(1 - Count).

Send_sr (Mbx_array(I), Buffer, Msg_size)

Send the message with size Msg_size which had been placed in Buffer to the mailbox referred to by Mbx_ array(I).

Receive_sr (Mbx_array(I), Buffer, Msg_size)

Wait for a message from the mailbox referred to by Mbx_array(I) to be read into Buffer , whose size is Msg_size.

UTILITIES

A set of utilities have been implemented to make Parallel Processing on the VAX system easier for the user. These utilities are really command files which have had symbols defined for them.

Cricomplink Program Compiler

This utility will compile and link Program using the compiler indicated by compiler, which may include optional parameters. It is helpful to use this utility because it handles special linking problems caused by shared memory access.

Crisetup Program Maxlog

This utility creates a set of command files for setting up the environment of this Program and also some debugging command files for use with the Parallel Debugger which assumes that the maximum number of logical processes used will be Maxlog. This needs to be executed only once for this Program unless the Parallel debugger is being used and Maxlog needs to be larger.

Crilogicals Program

This utility defines the logical names necessary to map to the shared memory. If this is not executed, local memory will be used exclusively.

Crisubmit Program
Logcpu Physcpu [After_time]

This utility starts up Program in Logcpu processes (logical cpus) on Physcpu processors (physical cpus) at time = After_time if present, otherwise now. In VMS terms, a command file which was setup for this program when Crisetup was executed, which in turn executes Crilogicals and then runs Program, is submitted to a generic batch queue Logcpu times. The generic queue will alternate the submittal amongst queues on Physcpu VAXs. Thus there will be Logcpu processes running, divided evenly amongst Physcpu processors.

Cricleanup

This utility needs to be executed only if there was an abnormal exit or the user wishes to abort the Job. Using a command file which was generated when Crisubmit was executed, it will remove any left over batch processes and delete shared memory access for the last Job submitted.

Cridebug Program Logcpu Physcpu

This utility starts up Program in Logcpu processes (logical cpus) on Physcpu processors (physical cpus) with the Parallel Debugger enabled.

IMPLEMENTATION

The implementation of the Parallel Processing Library on VMS was done using shared memory to store library information and interlocked instructions to update this information. Normally, temporary mailboxes in shared memory were used , which automatically go away when the Job completes. The shared memory must be mapped to a permanent global section and thus must be specifically deleted by the exit handler when the Job completes.

The root TASK of a parallel processing program is the program itself, all other TASKs are subroutines within that program. The same copy of the program is executed in all of the processes. The first process to execute the call to Task_init becomes the root TASK. The root TASK creates the necessary shared memory global sections , creates the Tasking mailbox and associates an exit handler for Job cleanup and termination. After the return from Task_init, it will continue executing the program. All other processes become slaves. A slave process also executes the call to Task_init, but after mapping to the shared memory global sections and to the Tasking mailbox which had been created by the root TASK, it will then perform a read on the Tasking mailbox and wait for a message. The slave processes will never proceed beyond the call to Task_init except to make subroutine calls requested by the Tasking mailbox message. Whenever Tskstart is called, a message is placed in the Tasking mailbox which indicates the subroutine Name to be called and its arguments. One and only one of the slave processes will receive that message; if a TASK is to be started, it will set up a taskblock for that TASK in shared memory, mark it valid, and then generate a call to that subroutine with the appropriate arguments. Upon returning from that subroutine, it will mark the TASK done. In order to wake up any other TASKs which might be waiting for this TASK, it will create a unique mailbox associated with that TASK, write to it, and then delete it. Having finished the business of that TASK, it will read the Tasking mailbox again to look for another TASK to do. When the process receives a DONE message in the Tasking mailbox, it will pass the message on and then commit suicide. The root TASK will automatically place the initial DONE message in the Tasking mailbox when it is finished by automatically using the exit handler that was set up by the call to Task_init.

The EVENT mechanism merely uses the VMS Common Event Flag clusters in shared memory.

The LOCK was implemented two ways. The first implementation did not care about the order in which the lock was granted. A LOCK was obtained by performing an interlocked decrement on the semaphor represented by Lockdata. If the LOCK was available, the process proceeded, otherwise it waited for an Event Flag which had been associated with that LOCK by the Lockasgn call. Lockoff did an interlocked increment on this semaphor and then set the Event Flag associated with this LOCK. Whichever TASK reacted the fastest got the LOCK next, there was no fairness criteria. This implementation appeared to be sufficient for a while. Later, a program, which used Locking in its Barrier implementation and synchronized on Barriers frequently, displayed very erratic behavior when executing on all 4 processors. This behavior was finally traced to a semi "starvation" effect caused by the unfair Locking mechanism. Processes were waiting excessively long within the Barrier due to lack of fair access to the LOCK which was used in that Barrier implementation. The Barrier was rewritten without the use of LOCKs and the erratic behavior disappeared. However, because of the possible "starvation" problem, it was decided to re-implement the Locking mechanism using an interlocked first in, first out (FIFO) queue. In addition to a semaphor, an interlocked queue was associated with each LOCK , both were represented by Lockdata. If a LOCK is not available, the Pid of the process requesting the LOCK and the nodename of its Processor are placed in the queue. When the LOCK becomes available, an entry is removed from the queue for that LOCK; if the waiting process resides on the same Processor, it is awakened. If the waiting process is on a different Processor from the process relinquishing the LOCK, a message is placed in a permanent shared mailbox associated

with that Processor. A server process responds to this message and wakes up the appropriate process on its Processor. The Program was retried with the old Barrier implementation and the new Lock implementation. The previous erratic behavior was not observed.

In order to allow processes to record their behavior in a synchronous manner, a set of subroutines was implemented which allows the user to easily write ordered records to a shared relative file; which may then be printed or otherwise interrogated. The last record used is noted in a shared memory array indexed into by the Unit number for that file.

To make dynamic load balancing easier, a set of subroutines was implemented for setting up, and reading and writing to shared mailboxes. The user must determine what information is necessary to indicate the next work item and must put that information into a buffer of appropriate size. After the mailbox has been setup, subsequent writes and reads to/from this mailbox will enter and remove items to/from the work queue represented by this mailbox.

RESULTS

Using the Parallel Processing Library described above, experiments were performed to investigate the benefits and the costs of parallel processing. For benchmarking purposes, two methods were used to implement Barriers (see Appendix A). These methods were implemented in assembly language in order to make them as fast as possible. Another method was implemented and tested using EVENTs and the Parallel Processing Library (see Appendix B); while using the other Barrier method, the elapsed time did not vary significantly from the first two Barrier methods. The first Barrier method, Method E, relinquishes the CPU (Processor) when it must wait at a Barrier and waits for an Event Flag associated with this Barrier to be set. The second Barrier method, Method S, spins , testing the shared memory location associated with this Barrier until it is ready. The difference in CPU time used by the two methods is the time that is spent waiting for the other Tasks to reach the Barrier.

The cost of using Barriers can be broken into components. There is the cost of the extra computations necessary to implement the Barrier; there is the cost of waiting within the Barrier due to resource contention, and there is the cost of waiting for the other TASKs to reach the Barrier. The last component can be estimated by using both kinds of Barriers and comparing the CPU times used. The second component is negligible for Method S , since the only resource contention present is a single interlocked decrement that occurs for each TASK when it first reaches the Barrier. Method E will have more resource contention due to its use of event flags. The first and second components were estimated by timing 100 loops of 60 consecutive Barrier calls, for Method E and for Method S. This test was run with from 1 to 4 TASKs, each with its own Processor. Method S took approximately .0001 seconds per Barrier, per TASK, no matter how many TASKs were running simultaneously. Method E took longer due to its use of event flags. As more TASKs participated, this become worse due to the added resource contention ; its time varied from .0006 seconds to .002 seconds depending on the number of TASKs participating.

Assuming a UNIT of COST to be the cost of a single +,-,* floating point type operation, the COST of a Lockon followed by a Lockoff, the COST of an Evpost followed by an Evclear, and the COST of the Barriers S and E were measured , varying the number of participating TASKs, each with its own Processor, from 1 to 4. See Table 1 for complete results. The COST for Locking varied from 9 to 65 units, depending on the number of TASKs, due to LOCK contention. The COST of Events varied less, from 33 to 46 units. The COST of Barrier E varied from 37 to 106 units, but the COST of Barrier S remained fairly constant at 7 units. Even though Barrier S appears to be cheaper, further results showed that the first two components of cost of a Barrier, which this test measures, are not the most important. Also, if the Processors are being shared, Barrier S would be wasting CPU cycles that others could be using. Another interesting side result of this experiment was that Barrier S, which uses shared memory heavily, did not degrade the performance of the System. This would seem to indicate that a potential hardware problem, shared memory contention, was not a problem in these experiments.

A standard LLNL benchmarking code named Simple, a hydrodynamic calculation with heat conduction, was used for further investigations. A grid size of 80 x 100 was used for 100 time cycles. There were 14 Barrier synchronizations performed per time cycle. Both Barrier methods (S,E) were used. The number of TASKs (Processors) working on the problem varied from 1 to 4. The number of Logical Processors actually working on the problem was never greater than the number of Physical Processors. No other users were on the System during benchmarks. At each Barrier call, for each TASK, data was saved indicating when the Barrier was entered and when it was exited. Upon termination of the Job, this data was processed . All the Barrier delays were summed and averaged among the number of TASKs (WTave). Also, at each Barrier, the maximum delay amongst the participating Tasks was found, and these were summed (WTM). From the logs, the CPU usage for Barrier E was subtracted from the CPU usage for Barrier S and the difference was divided by the number of participating Tasks to give the average wait at a Barrier (Wave). With complete parallelism, if one TASK takes X seconds to complete the Job, then N TASKs should take X/N seconds. If T is the time that it actually took to run the Job with N TASKs, then let D be the Discrepancy, where D = (T - X/N). The speedup is usually a measure of how much parallelism was actually achieved. Speedup(N) = Elapsed time for one TASK/ Elapsed time for N TASKs.

The first experiments were done using fixed partitioning of the work load. The work was divided up equally amongst the Tasks before starting the Job. The time for 1 Task to complete the Job was 1600 seconds. A Job was run using 4 Tasks and bypassing the synchronization; the answers were wrong, but the Speedup was = 4! Using the Barrier synchronizations, Speedup(2) = 1.95, Speedup(3) =

2.85, Speedup(4) = 3.7. The Speedup did not vary significantly as a function of the Barrier implementation method used, including the one in Appendix B. For complete results, see Table 2. The total cost of the Barriers (first and second components) of this Job is approximately = cost of a single Barrier X 14 X 100. Therefore, Barrier E cost from .8 to 2.8 sec depending on the number of Tasks. Barrier S cost approximately .14 sec. In any case, the cost is < 1% of the total cost of the Job. Then, why isn't the Speedup better? It appears that the third component of the Barrier cost, the wait at the Barriers for the other TASKs, is the primary expense in Barrier synchronization for this Job. Even if this Job has exclusive use of the System, it still does not have exclusive use of the Processors. The Operating System must continue to do its work (cluster management, accounting, error logging, etc). Bare in mind that there is not 1 Operating System, but rather 4 Operating Systems involved. If any of these Operating System uses CPU cycles, the TASK being run under that Operating System will be delayed, and all other TASKs will have to wait for the delayed TASK when a Barrier is encountered. Figure 1 shows a scatter plot showing the distribution of the sizes of the maximum waits at the Barriers, looking at all 14 barriers, but only 20 cycles worth of data. Figure 2 shows a plot of how the size of the maximum wait at a single Barrier varies, using the same 20 cycles worth of data.

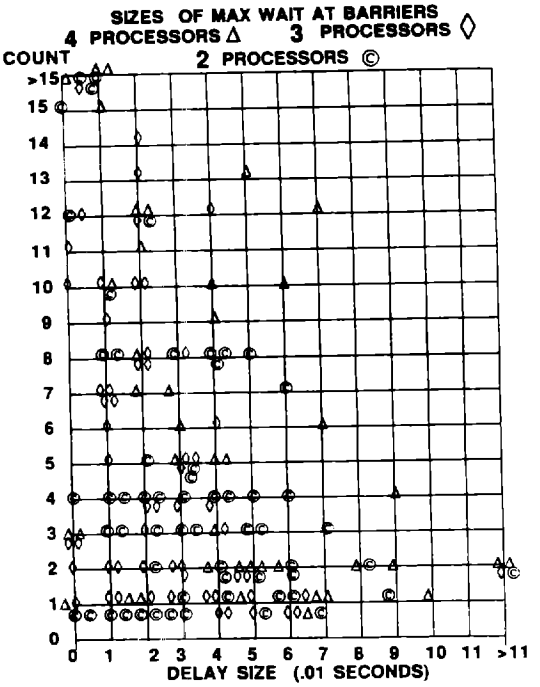


figure 1

Table 1

COST OF PARALLEL PROCESSING

Definition: UNIT of COST = one *,-,+ Operation

FUNCTION	COST (1 CPU)	COST (2 CPUS)	COST (3 CPUS)	COST (4 CPUS)
LOCK (ON/OFF)	9	10	42	65
EVENT (POST/CLEAR)	33	43	46	44
BARRIER (E)	37	78	92	106
BARRIER (S)	6	7	7	7

If the Barriers are few, with a large amount of work being done in between, these delays might statistically even out amongst the TASKs, causing less delay at the Barriers due to waiting for each other. In other words, the larger the granularity of the problem between Barriers, the more efficient use the Job will make of the Processors.

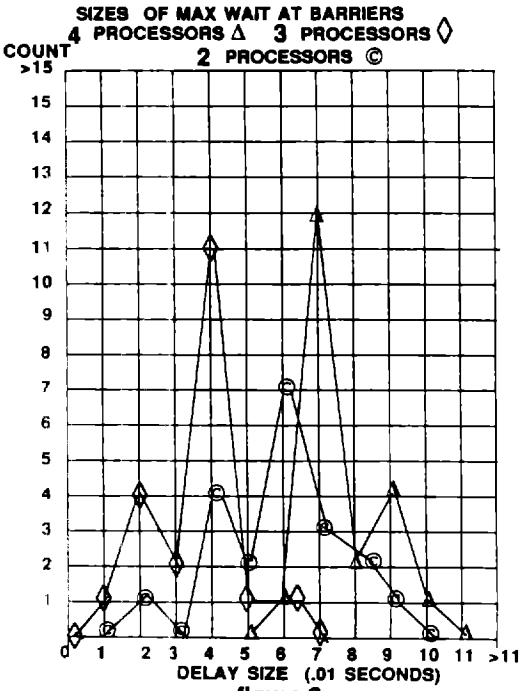


figure 2

Table 2

SIMPLE WITH FIXED EQUAL PARTITIONS

#PROCESSORS	SPEEDUP	ELAPSED TIME	D Wave	WTave	WTM
1	1.00	1600	00 00	00	00
2	1.95	820	20 15	20	38
3	2.85	560	27 13	15	30
4	3.70	430	30 16	21	45
4	4.00	NO SYNCHRONIZATION			

Work partitioning was investigated next. The work was divided up into Work Queues with a fixed number of items in each queue. Each TASK is allowed to remove items from the queue until that work is completed, at which time a Barrier is usually encountered. As items are removed from one Work Queue and worked on, they are generally inserted into the next work queue. It was found that the overhead cost of using Work Queues was approximately .003 seconds or 120 COST UNITS per Work Item, per Barrier (see Table 3).

Table 3

SIMPLE WITH DYNAMIC PARTITIONING

OVERHEAD OF WORK QUEUES (1 PROCESSOR)

# WORK ITEMS	ADDED ELAPSED TIME (seconds)
01	000
20	080
40	170
60	250
80	350

OVERHEAD approximately = 4 seconds per Work Item, per Job
= .003 seconds or 120 Cost UNITS
per Work Item, per Barrier

If a Job takes 1600 seconds for 1 Processor to complete, the best that can be done with 4 Processors would be 400 seconds. The Overhead can be estimated to be the Actual CPU Usage per Processor - Best Possible CPU Usage per Processor. The Delay at the Barriers is again estimated by comparing the CPU usage of the two different Barrier implementations. As we can see from Figure 3, as the number of work items in the Work Queue increases, the delay at the Barriers tends to go down, but the overhead goes up. In fact, from Figure 4, it is evident that for this Job, the Fixed Partitioning Method is superior to the Dynamic Partitioning Method. The Overhead of Dynamic Partitioning exceeds any Delays caused by the work load imbalance of this Job.

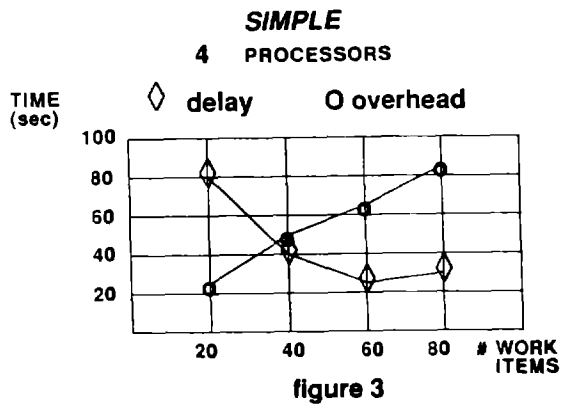


figure 3

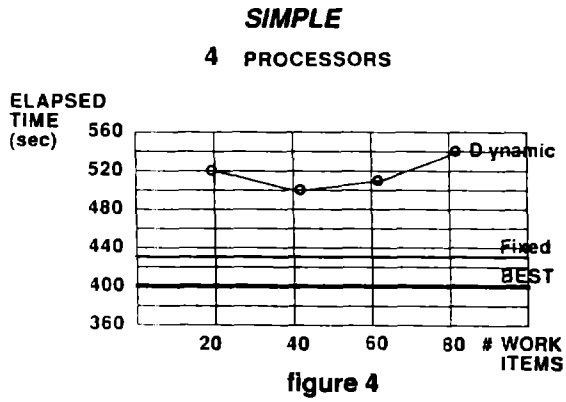


figure 4

The expected Elapsed Time for a Job can be estimated to be the average CPU used (per Processor) plus the average Delay at the Barriers. We can then guess the CPU utilization for this Job when it is executing to be the Estimated Elapsed Time divided by the Actual Elapsed Time. These results are shown in Table 4. The Job was monitored during its execution and the estimated CPU utilization numbers were actually observed to be true. It is assumed that not only was the Job doing more work when using Work Queues, but the Operating System was also doing more work. Even though Work Queues were not the most efficient implementation of this Job in an exclusive environment, that does not mean that they won't be for another Job which has to deal with greater load imbalance problems or even ,perhaps, has to share the system with "other" users!

Table 4

SIMPLE, FIXED EQUAL PARTITIONING VERSUS DYNAMIC PARTITIONING

BARRIER METHOD	# WORK ITEMS	ACTUAL ELAPSED TIME	CPU AVE	OVERHEAD	Wave	WTave	WTM	EST. ELAPSED TIME	%CPU UTI.
FIXED		430	405	05	16	21	45	421	98
DYNAMIC	20	520	425	25	80	80	200	505	97
	40	500	445	45	40	40	115	485	97
	60	510	465	65	25	25	55	490	96
	80	540	485	85	30	30	100	515	95

SUMMARY

Many things were learned from these parallel processing experiments. Mostly, it was learned that parallel processing is not easy. The difficulties in constructing a "correct" program and knowing that it is indeed "correct" were not even discussed. Some factors that might influence implementation techniques were explored. The efficiency of these

techniques depends not only on the problem being solved, but on the arch tecture of the computer being used to solve it. To avoid unnecessary overhead and delays, synchronization should be minimized whenever possible. New mathematical algorithms need to be designed with this in mind. General schemes for solving parallel processing problems will need to be modified to suit each parallel processing environment. At present, the programmer is almost totally responsible for finding and explicitly declaring the parallel processing capabilities of his Job. Eventually compilers will assist, if not relieve, the programmer of that responsibility. There are still many unknowns concerning the suitability of computer architectures, computer algorithms, and computer software for solving the problems inherent in parallel processing. Experimenting with parallel processing will give some useful insights into the problem.

REFERENCES

[1] "Multitasking User's Guide'", Cray Research, Inc., Mendota Heights, MN Sn-0222

[2] Werner, N.E., Van Matre, S.W., "Parallel Processing on the Livermore VAX 11/780-4 Parallel Processor System with Compatibility to Cray Research, Inc. (CRI) Multitasking", Version 1, UCRL-92624, May 1985

BARRIER METHOD E

DATA:

3 pairs of integers, ordered,
<THIS,NEXT,LAST>
Synchronization_variable
Event_flag_number

Number_of_TASKs = number of TASKs
synchronizing

INITIALIZATION:

Set Synchronization_variable(s) to
Number_of_TASKs
Clear event flags = Event_flag_number(s)
Initialize the order of the data items
THIS = now
NEXT = next to be used
LAST = last one used before now

SYNCHRONIZE:

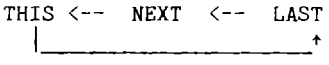
Decrement (interlocked)
THIS Synchronization_variable

If THIS Synchronization_variable not = 0,
then wait for THIS Event_flag_number

Otherwise Post THIS Event_flag_number

RESET:

Set LAST Synchronization_variable to
Number_of_TASKs
Clear LAST Event_flag_number
Rotate the order of the Data Items



BARRIER METHOD S

DATA:

3 ordered integers, <THIS,NEXT,LAST>

Number_of_TASKs = number of TASKs
synchronizing

INITIALIZATION:

Set Synchronization_variable(s) to
Number_of_TASKs
Initialize the order of the data items
THIS = now
NEXT = next to be used
LAST = last one used before now

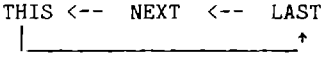
SYNCHRONIZE:

Decrement (interlocked) THIS
Synchronization_variable

Test THIS Synchronization_variable
until THIS Synchronization_variable = 0

RESET:

Set LAST Synchronization_variable to
Number_of_TASKs
Rotate the order of the Data Items



Appendix B

BARRIER METHOD USING
THE PARALLEL PROCESSING LIBRARY

```
DATA:

C       Array of 3 ordered event flag numbers for
C       each participating TASK

        Integer Event_numbers( 3, Number_of_TASKs )

        Integer       THIS,NEXT,LAST

        Integer       Number_of_TASKs, Task_id
        Number_of_TASKs = number of Tasks
                                synchronizing

        Task_id =  This TASK's identification number
        where  0  <  Task_id  <  Number_of_TASKs + 1


INITIALIZATION:

C       Clear this TASK's event flags

        DO I = 1, 3

        Call Evclear ( Event_numbers(I, Task_id))

        End do


C       Initialize the order of the data items


             THIS = 0  ;  use now
             NEXT = 1  ;  next to be used
             LAST = 2  ; last one used before now


SYCHRONIZE:

C       Signal that this Task is ready

        Call Evpost  ( THIS, Task_id )


C       Wait for all Tasks

        Do I = 1, Number_of_TASKs
        Call Evwait ( THIS, I)
        End Do


RESET:

C       Reset appropriate last signal

        Call Evclear ( LAST, Task_id )

C       Rotate Data Items

        LAST = THIS
        THIS = IMod (THIS + 1, 3)
        NEXT = IMod (THIS + 1, 3)


        Return
```